DeepCPU: Serving RNN-based Deep Learning Models 10x Faster

Minjia Zhang, Samyam Rajbhandari, Wenhan Wang,
and Yuxiong He, *Microsoft AI and Research*

https://www.usenix.org/conference/atc18/presentation/zhang-minjia

This paper is included in the Proceedings of the
2018 USENIX Annual Technical Conference (USENIX ATC '18).
July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the
2018 USENIX Annual Technical Conference
is sponsored by USENIX.

DeepCPU: Serving RNN-based Deep Learning Models 10x Faster

Minjia Zhang* Samyam Rajbhandari* Wenhan Wang Yuxiong He
Microsoft Business AI and Research
{*minjiaz,samyamr,wenhanw,yuxhe*}@*microsoft.com*

Abstract

Recurrent neural networks (RNNs) are an important class of deep learning (DL) models. Existing DL frameworks have unsatisfying performance for online serving: many RNN models suffer from long serving latency and high cost, preventing their deployment in production.

This work characterizes RNN performance and identifies low data reuse as a root cause. We develop novel techniques and an efficient search strategy to squeeze more data reuse out of this intrinsically challenging workload. We build DeepCPU, a fast serving library on CPUs, to integrate these optimizations for efficient RNN computation. Our evaluation on various RNN models shows that DeepCPU improves latency and efficiency by an order of magnitude on CPUs compared with existing DL frameworks such as TensorFlow. It also empowers CPUs to beat GPUs on RNN serving. In production services of Microsoft, DeepCPU transforms many models from non-shippable (due to latency SLA violation) to shippable (well-fitting latency requirements) and saves millions of dollars of infrastructure costs.

1. Introduction

Deep learning (DL) is a fast-growing field pervasively influencing many applications on image, speech, and text processing. Traditional feed forward neural networks assume that all inputs (and outputs) are independent of each other. This could be a bad idea for many tasks. For example, to predict the next word in a sentence, we had better know which words come before that. To classify what kind of event is happening to the next point of a movie, we had better reason from the previous events. Recurrent neural networks (RNNs) are an important and popular class of DL models that address this issue by making use of sequential information [22,35,51]. RNNs perform the same task for every element in the sequence, with the output being dependent on the previous computation. This is somewhat similar to the human learning, e.g., to understand a document, we read word by word, sentence by sentence, and carry the information along in our memory while reading. RNNs have shown great promise in many natural language processing tasks, e.g., language model [16,44], machine translation [15,21,58], machine reading comprehension [18,25,39,53], speech recognition [31,34,66], and conversational bots [62].

Like other DL models, using RNNs requires two steps: (1) learning model weights through training, and (2) applying the model to predict the results of new requests, which is referred to as *serving*, or equivalently, inferencing or scoring. Training is a throughput-oriented task: existing systems batch the computation of multiple training inputs to obtain massive parallelism, leveraging GPUs to obtain high throughput. Users can often tolerate fairly long training time of hours and days because it is offline. Serving, on the other hand, makes online prediction of incoming requests, imposing different goals and unique challenges, which is the focus of this paper.

Latency and efficiency are the two most important metrics for serving. Interactive services often require responses to be returned within *a few or tens of milliseconds* because delayed responses could degrade user satisfaction and affect revenue [27]. Moreover, large-scale services handle massive request volumes and could require thousands of machines to serve a single model. Many RNN models from production services such as web search, advertisement, and conversational bots require intensive computation and could not be shipped because of serving latency violation and cost constraints.

Detailed investigation shows that popular DL frameworks, e.g., TensorFlow and CNTK, exhibit poor performance when serving RNNs. Consider the performance metric of floating point operations per second (flops), which is a standard measure for computations like DL that are dominated by floating-point calculations. Our test results show that on a modern Intel CPU with peak performance of 1.69Tflops, using TensorFlow/CNTK for RNN serving only gets less than 2% of hardware peak. This naturally raises many questions: Why is there such a big performance gap between hardware peak and the existing implementations? Are we dealing with an intrinsically challenging workload or less optimized systems? Would different hardware, such as GPU, help?

We carefully characterize RNN performance and answer the above questions.

First, RNN serving is an intrinsically challenging workload. Due to stringent latency SLA, online serving systems often process each request upon its arrival, or at best, batch a few requests whenever possible. With a batch size of 1 (or a few), the computation is dominated

*Both authors contributed equally. Order of appearance is random.

by several vector-matrix multiplications (or matrix multiplications), that have poor *data reuse* and thus are bottlenecked on cache/memory bandwidth. Since the speed of data transfer is far slower than the computational speed of CPUs, this leaves cores waiting for data instead of conducting useful computation, leading to poor performance and latency.

Second, existing DL frameworks rely on parallel-GEMM (GEneral Matrix to Matrix Multiplication), implementations which are not targeted to optimize the type of matrix multiplications (MMs) in RNN computations. parallel-GEMM is designed to optimize large MMs with high data reuse by hiding the data movement cost with ample computation [29]. MMs in RNNs are usually much smaller, fitting entirely in shared $L3$ cache, but with minimal data reuse: data movement from shared $L3$ cache to private $L2$ cache is the main bottleneck. Due to limited data reuse, parallel-GEMM can no longer hide the data movement, requiring different considerations and new techniques. Furthermore, as weights are repeatedly used at MMs of each step along the sequence, it presents a potential reuse opportunity from RNN domain knowledge, which parallel-GEMM does not exploit.

Lastly, would GPU help? RNN serving is computationally intensive but with limited parallelism. In particular, the amount of computation grows linearly with the sequence length: the longer the sequence, the more steps the computation carries. However, the sequential dependencies make it hard to parallelize across steps. As the batch size is also small in serving scenario, there is rather limited parallelism for RNN serving. As GPUs use a large number of relatively slow cores, they are not good candidates because most of the cores would be idle under limited parallelism; CPUs are a better fit with a smaller number but faster cores.

With the challenges and opportunities in mind, we develop novel techniques to optimize data reuse. We build DeepCPU, an efficient RNN serving library on CPUs, incorporating the optimization techniques. Our key techniques include (1) private-cache-aware partitioning, that provides a principled method to optimize the data movement between the shared $L3$ cache to private $L2$ cache with formal analysis; (2) weight-centric streamlining, that moves computation to where weights are stored to maximize data reuse across multiple steps of RNN execution. Both help overcome the limitation of directly applying parallel-GEMM and optimize data reuse on multi-core systems. We also leverage existing techniques, such as MM fusion and reuse-aware parallelism decision, in the new context of RNN optimization.

Effectively integrating these techniques together is non-trivial, requiring to search a large space to find optimized schedules. We model RNN computation using a Directed Acyclic Graph of Matrix Multiplication nodes (MM-DAG), supporting a rich set of optimization knobs such as partitioning (splitting a node) and fusion (merging nodes). It is well known that the traditional DAG scheduling problem of minimizing execution time by deciding the execution order of the nodes is NP-hard even in the absence of additional knobs [28]. The optimization knobs further enlarge the search space exponentially, and it is infeasible to exhaustively enumerate all schedules. We develop an efficient search strategy that requires far fewer calibration runs.

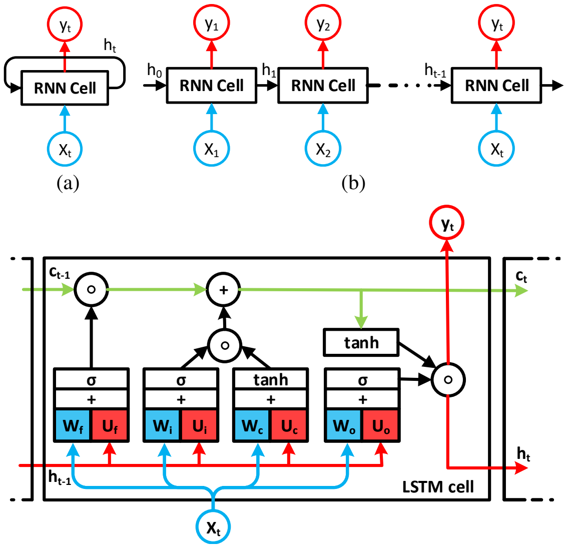
We compare DeepCPU with popular state-of-the-art DL frameworks, including TensorFlow and CNTK, for a wide range of RNN models and settings. The results show DeepCPU consistently outperforms them on CPUs, improving latency by an order of magnitude. DeepCPU also empowers CPUs to beat highly optimized implementations on GPUs. We further demonstrate its impact on three real-world applications. DeepCPU reduces their latency by 10–20 times in comparison to TensorFlow. To meet latency SLA, DeepCPU improves the throughput of the text similarity model by more than 60 times, serving the same load using less than 2% of machines needed by the existing frameworks.

The key contributions of the work include: 1) Characterizing performance limitations of the existing methods (Section 3). 2) Developing novel techniques and a search strategy to optimize data reuse (Section 4 and 5). 3) Building DeepCPU, a fast and efficient serving library on CPUs (Section 4 and 5). 4) Evaluating DeepCPU and showing order of magnitude latency and efficiency improvement against the existing systems (Section 6).

DeepCPU has been extensively used in the production of Microsoft to reduce serving latency and cost. It transforms the status of many DL models from impossible to ship due to violation of latency SLA to well-fitting SLA requirements. It empowers bigger and more advanced models, improving accuracy and relevance of applications. DeepCPU also greatly improves serving efficiency, saving thousands of machines and millions of dollars per year for our large-scale model deployments.

2. Background

An RNN models the relationships along a sequence by tracking states between its steps. At each step t (Fig. 1a), it takes one unit of input x_t (e.g., a token in a text, or a phoneme in a speech stream) and makes a prediction y_t based on both the current input x_t and the previous hidden (or cell) state h_{t-1}. The hidden states $\{h_t\}$ form a loop, allowing information to be passed from one step to the next. The block of computation per step is called an RNN cell, and the same cell computation is used for all inputs of the sequence. An RNN (sequence) computation can be viewed as an unrolled chain of cells (Fig. 1b).



(a)

(b)

(c)

Figure 1: a) RNN with a recurrent structure. b) Unrolled RNN. c) LSTM structure.

LSTM/GRU. There are many variations of RNNs, inheriting the recurrent structure as above but using different cell computations. The two most popular ones are Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU) network, best known for effectively catching long-term dependencies along sequences. We use LSTM as an example and illustrate its cell computation:

$$\begin{aligned}
i_t &= \sigma(\mathbf{W}_i \cdot x_t + \mathbf{U}_i \cdot h_{t-1} + b_i) \\
f_t &= \sigma(\mathbf{W}_f \cdot x_t + \mathbf{U}_f \cdot h_{t-1} + b_f) \\
o_t &= \sigma(\mathbf{W}_o \cdot x_t + \mathbf{U}_o \cdot h_{t-1} + b_o) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tanh(\mathbf{W}_c \cdot x_t + \mathbf{U}_c \cdot h_{t-1} + b_c) \\
h_t &= o_t \circ \tanh(c_t) \ .
\end{aligned}$$

Here $\sigma(\cdot)$ denotes the sigmoid function. Online tutorials [7, 12] describe good insights of the formulation on how it facilitates learning. Here we focus on describing the main computations. We denote E as the input dimension of the input vector x_t, and H as the hidden dimension of the hidden vector h_t. LSTM includes 4 input MMs, which multiply input vector x_t with four input weight matrices $\mathbf{W}_{\{i,f,o,c\}}$ of size $E \times H$ each (marked as blue in Fig. 1c). It has 4 hidden MMs, which multiply hidden vector h_{t-1} with four hidden weight matrices $\mathbf{U}_{\{i,f,o,c\}}$ of size $H \times H$ each (red in Fig. 1c). Within each cell, there is no dependency among the 8 MMs, and across cells, the hidden state of step t depends on step $t-1$ (as shown by Fig. 1c). LSTM also consists of a few element-wise additions ($+$) and products ($\circ$), as well as activation functions such as σ and tanh.

Similar to the LSTM cell, GRU cell has 6 instead of 8 MMs but with additional dependencies within them [22].

Single vs. batch mode. To make real-time predictions, online requests are often processed one by one as they arrive, or occasionally, under a small batch. Given a batch size of B, the batched input x_t can be represented as a matrix of size $B \times E$, which transforms the underlying computation from a vector-matrix to a matrix-matrix multiplication, exposing more opportunities for data reuse. However, because of tight latency requirements and spontaneous request arrivals, the batch size at serving is usually much smaller (e.g., 1 to 10) than the large mini-batch size (often hundreds) during training.

3. Performance Characterization

Existing DL frameworks such as TensorFlow/CNTK implement RNNs as a loop of cell computation: as shown in Lis. 1, 8 MMs in the LSTM cell are fused into a single MM, executed using parallel BLAS libraries such as Intel-MKL [3], OpenBLAS [6] or Eigen [1]. We measure their performance in serving scenarios with small batch size from 1 to 10. On a dual-socket Xeon E5-2650 CPU machine, we often observe performance of $<$ 30Gflops: less than 2% of the machine peak of 1.69Tflops. What is the cause of such a big gap?

Listing 1: LSTM Implementation in TensorFlow/CNTK

```
for $t$ in input_sequence:
    $[f'_t \ \ i'_t \ \ o'_t \ \ c'_t] = [x_t \ \ h_{t-1}] \begin{bmatrix} \mathbf{W}_f & \mathbf{W}_i & \mathbf{W}_o & \mathbf{W}_c \\ \mathbf{U}_f & \mathbf{U}_i & \mathbf{U}_o & \mathbf{U}_c \end{bmatrix}$
    $c_t = \sigma(f'_t) \circ c_{t-1} + \sigma(i'_t) \circ \tanh(c'_t)$
    $h_t = \sigma(o'_t) \circ \tanh(c_t)$
```

The first step of performance analysis is to identify the dominating computation. In RNNs, the total amount of computation is dominated by MMs. The total ops in MMs per RNN cell are $\mathcal{O}(B \times (E+H) \times H)$, and the total ops in element-wise operations and activations functions are $\mathcal{O}(B \times H)$. Typically the total number of ops in MMs is two to three orders of magnitude larger than the rest combined. As such, RNN performance primarily depends on the MMs, which is the focus of this study.

We analyzed MMs in the RNNs, and identified three key factors causing poor performance.

i) Poor data reuse. Data reuse at a particular level of memory hierarchy is a measure of the number of computational ops that can be executed per data load/store at that level of memory hierarchy. Assuming a complete overlap between computation and data movement (best case scenario), the execution time of a computation can be estimated as a function of the data reuse using the roofline model [65] as

$$\begin{aligned}
Time &\geq Max(DataMoveTime, CompTime) \quad (1) \\
&= Max(\frac{DataMoved}{DataBandwidth}, \frac{TotalComp}{Peak}) \\
&= Max(\frac{TotalComp/Reuse}{DataBandwidth}, \frac{TotalComp}{Peak})
\end{aligned}$$

Based on this execution time, note that poor data reuse results in poor performance because on modern architectures, the computational throughput is significantly higher than the data movement throughput. Let us look at an example of $L3$ to $L2$ bandwidth since all RNN models we have seen fit in $L3$ cache of modern CPUs: the *peak* computational performance of a Xeon E5-2650 machine is 1.69Tflops while the observable *DataBandwidth* between $L3$ and $L2$ cache on it is 62.5 GigaFloats/s (250 GB/s), measured using the stream benchmark [8]. If the reuse is low, the total execution time is dominated by the data movement, resulting in poor performance.

This is indeed the case for RNN in serving scenario where the batch size tends to be very small. To see this, consider an MM:$\mathbf{C}[i,j] = \sum_k \mathbf{A}[i,k] \times \mathbf{B}[k,j]$. If we assume that both the inputs and the outputs reside in $L3$ cache at the beginning of the computation, then both the inputs and the outputs must be read from $L3$ cache to $L2$ cache at least once, and the outputs must be stored from $L2$ cache to $L3$ cache at least once during the MM. Therefore, the maximum possible data reuse during this MM from $L2$ cache is given by $\frac{2\times I\times J\times K}{|\mathbf{A}|+|\mathbf{B}|+2|\mathbf{C}|}$, where I,J and K are the size of indices i,j and k. Similarly, the fused MM of LSTM has the shape $[B,E+H]\times[E+H,4H]$, and its data reuse is:

$$MaxDataReuse = \frac{8\times B\times H\times(E+H)}{|Input|+|Weights|+2|Output|} \quad (2)$$

$$= \frac{8\times B\times H\times(E+H)}{B\times(E+H)+4\times(E+H)\times H+8\times B\times H} \quad (3)$$

When batch size $B \ll \min(H,E)$, the maximum data reuse in Eqn. 2 reduces to $2B$. Take $B = 1$ as an example: the best achievable performance of LSTM on the Xeon E5-2650 machine is at most 125Gflops based on the measured $L3$ bandwidth of 250 GB/s. This is less than 8% percent of the machine's peak of 1.69Tflops.

ii) Sub-optimal MM partitioning. Parallel-GEMM libraries are designed to optimize performance of large MMs that have significant data reuse (> 1000). They exploit this reuse from $L2$ cache level using loop-tiling to hide the data movement cost from both memory and $L3$ cache [29]. In contrast, the amount of reuse in RNNs is in the order of B, which is often a small value between 1 and 10 for most serving cases. This is not enough to hide the data movement cost even though MMs in RNN are small enough to fit in $L3$ cache. In the absence of large reuse, the performance of parallel-GEMM is limited by the data movement cost between shared $L3$ cache and private $L2$ caches. Parallel-GEMM is sub-optimal at minimizing this data movement.

More specifically, $L3$ cache on a modern CPU feeds to *multiple* $L2$ caches that are private to each core. During RNN computations, some data might be required by multiple cores, causing multiple transfers of the same piece of data from $L3$ cache. Thus, the total data movement between $L3$ and $L2$ caches depends on the partitioning of the MM computation space and its mapping to the cores. For example, if we split an MM computation among two cores, such that the first core computes the upper half of the output matrix $\mathbf{C}$, while the second core computes the lower half, then input matrix $\mathbf{B}$ must be replicated on $L2$ cache of both cores, as the entire matrix $\mathbf{B}$ is required to compute both halves of matrix $\mathbf{C}$. Alternatively, if the computation is split horizontally, then the input matrix A must be replicated on $L2$ cache of both cores. Different partitionings clearly result in different amount of data reuse. Parallel-GEMM does not always produce a partitioning that maximizes this data reuse. Libraries specialized for small matrices are not sufficient either, as some focus only on sequential execution [56] while others focus on MM small enough to fit in $L1$ cache [43].

iii) No data reuse across the sequence. During serving, weight matrices of RNNs remain the same across the sequence, but existing solutions do not take advantage of that to optimize data reuse. More precisely, parallel-GEMM used to execute the MMs is not aware of this reuse across the sequence. During each step of the sequence, the weight matrix could be loaded from $L3$ cache to $L2$ cache. However, it is possible to improve performance of RNNs by exploiting this data reuse.

Beyond MM: Beyond limited data reuse at MMs, existing RNN implementations in DL frameworks such as TensorFlow have other performance limiting factors, e.g., data transfer overheads among operators, buffer management overheads, unoptimized activation functions, which we address in DeepCPU. For example, we develop efficient SIMD implementations of *tanh* and *sigmoid* activation functions using continued fraction expansion, supporting any desired degree of precision by adjusting the number of terms to terminate the expansion [60]. Since these improvements mostly require good engineering practice than novel methods, we did not discuss them in detail for the interest of space.

4. Challenges and Strategies

Challenges. Finding an optimized implementation for RNN execution that maximizes data reuse while also efficiently using low-level hardware resources (such as SIMD hardware) is challenging due to the explosive space of optimization knobs and execution schedules. Practically infinite number of valid choices can be obtained through loop permutations, loop fusions, loop unrolling, unroll factor selection, loop tiling, tile-size selection, MM reordering, vectorization, register tiling, register tile size selection, parallel loop selection, parallelization granularity selection, thread-to-core mapping etc., and their combinations. Furthermore, enabling those optimization knobs and creating a schedule generator for all choices is a non-trivial engineering task. Additionally, the optimal choice is dependent on both hardware architecture and RNN parameters: a single solution will not work for all cases and an optimized schedule needs to be tuned case by case in an efficient manner. All of the above make the problem challenging in practice.

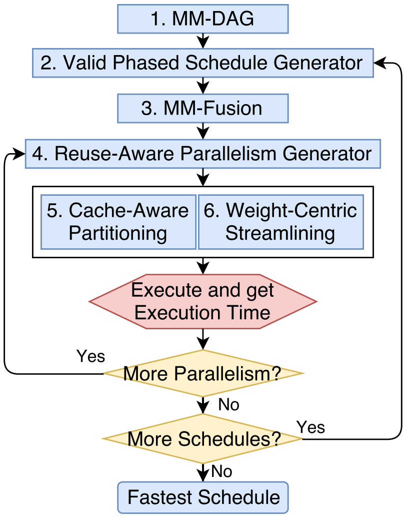

1. MM-DAG
2. Valid Phased Schedule Generator
3. MM-Fusion
4. Reuse-Aware Parallelism Generator
5. Cache-Aware Partitioning
6. Weight-Centric Streamlining
Execute and get Execution Time
Yes
More Parallelism?
No
More Schedules?
Yes
No
Fastest Schedule

Figure 2: DeepCPU optimization overview.

Strategies: DeepCPU overview. To overcome these challenges, we judiciously define the search space and identify the most important techniques to boost data locality. This empowers efficient search within a selective set of optimization knobs and schedules for obtaining the best RNN performance. We build the entire optimization pipeline into a library, which we call DeepCPU. Fig. 2 highlights its key features and workflow.

An important start of the optimization is to define a concise search space, which we develop upon two insights. (1) We identify the most performance critical operators, MMs, and model the computation graph connecting them to capture the first-order impact. We can do this by constructing a Matrix Multiplication Directed Acyclic Graph (MM-DAG) to represent the RNN computation, where each node represents an MM and edges represent dependencies among them. This model allows us to build schedules using MMs as the basic building blocks, capturing key computations while abstracting away other low-level details. (2) Instead of examining all valid schedules for the MM-DAG, which is not trackable, we can leverage the iterative nature and other properties of RNNs, prune search space to deduplicate the performance-equivalent schedules, and remove those that cannot be optimal. These two insights are implemented as $\boxed{1}$ and $\boxed{2}$ in Fig. 2.

We then identify and develop four techniques to effectively boost data locality for RNNs, applying them on each schedule (shown as $\boxed{3}$,$\boxed{4}$,$\boxed{5}$,$\boxed{6}$ in Fig. 2):

- **MM-fusion:** fuses smaller MMs into larger ones, improving data reuse;
- **Reuse-aware parallelism generator:** identifies best parallelism degree within and across MMs through auto-tuning, jointly considering locality;
- **Private-cache-aware-partitioning (PCP):** optimizes data movement between shared $L3$ cache and private $L2$ cache with a novel and principled partitioning method;
- **Weight centric streamlining (WCS):** maps the partitions produced by PCP to cores in a way that enables reuse of weights across the sequence.

The parallelism generator $\boxed{4}$ iterates over different choices on parallelism degrees. For a parallelism choice, we use PCP $\boxed{5}$ to obtain locality optimized parallel partitions. The partitions are then mapped to cores using WCS $\boxed{6}$. Individual partitions are implemented using highly optimized single-threaded BLAS library which optimizes for low-level hardware resources such as $L1$ cache and SIMD instruction set. DeepCPU applies this schedule to obtain the execution time, and loop over to find the best parallelism choice. Once this process is completed for all schedules generated by $\boxed{2}$, DeepCPU simply chooses the schedule that is the fastest. This calibration process is often called once during model construction, and then the optimized schedule is repeatedly used for serving user requests of the model.

In the design of DeepCPU, we deliberately combine analytical performance analysis (at search space pruning and PCP) with empirical calibration (to measure the combined impact of locality and parallelism). The former effectively reduces the search space, saving tuning time to run many suboptimal/redundant schedules. The latter reliably measures the actual execution time to capture complex software and hardware interaction, which can hardly be accurately estimated. This combination empowers both effectiveness and efficiency.

5. DeepCPU Optimizations

This section dives into DeepCPU optimizations from refining search space to locality optimizations. We conclude it by demonstrating the performance breakdown and impact of these optimizations.

5.1. MM-DAG Scheduling

DeepCPU models RNN computations as MM-DAGs and optimizes the schedules to execute them. Given an MM-DAG, a valid schedule determines an execution ordering of its nodes that satisfies all the dependencies. We consider only those valid schedules that are composed of phases: A *phased schedule* executes an MM-DAG in a sequence of phases $S_1, S_2, S_3, ..., S_i, ...$, where each phase S_i represents a non-overlapping subset of nodes and $S = \sum_i S_i$ consists of all nodes. There is a total ordering between phases such that if $i < j$, then all nodes in S_i must be executed before S_j. However, nodes within a phase can be executed in parallel. Lst. 2 shows two examples of valid phased schedules for LSTM. In Schedule 1, all MMs at a timestep t are in Phase t.

The phases can be divided into two categories: i) If a phase consists of an MM that has dependency across the timesteps, we call it a *time-dependent phase*, e.g., those MMs taking hidden state h_t as inputs, ii) Otherwise, if a phase does not contain any MM that has dependency across the sequence, we call it a *time-independent phase*. For example, in Schedule 2 of Lst. 2, Phase 1 is time-independent, and consists of all the MMs computing input transformation (with weights $\mathbf{W_i}, \mathbf{W_f}, \mathbf{W_c}$ and $\mathbf{W_o}$) across all timesteps; all other phases are time-dependent, requiring the value of h_{t-1} to compute h_t.

Listing 2: Phased LSTM Schedule-1 and 2

```
//Phased LSTM Schedule 1
for t:
  Phase t:    //time-dependent
    $\mathbf{W}_i \cdot x_t, \mathbf{W}_f \cdot x_t, \mathbf{W}_c \cdot x_t, \mathbf{W}_o \cdot x_t$
    $\mathbf{U}_i \cdot h_{t-1}, \mathbf{U}_f \cdot h_{t-1}, \mathbf{U}_c \cdot h_{t-1}, \mathbf{U}_o \cdot h_{t-1}$

//Phased LSTM Schedule 2
Phase 1:    //time-independent
   $\mathbf{W}_i \cdot x_0, .., \mathbf{W}_i \cdot x_t, \mathbf{W}_f \cdot x_0, .., \mathbf{W}_f \cdot x_t,$
   $\mathbf{W}_c \cdot x_0, .., \mathbf{W}_c \cdot x_t, \mathbf{W}_o \cdot x_0, .., \mathbf{W}_o \cdot x_t$
for t:
  Phase (t+1):  //time-dependent
     $\mathbf{U}_i \cdot h_{t-1}, \mathbf{U}_f \cdot h_{t-1}, \mathbf{U}_c \cdot h_{t-1}, \mathbf{U}_o \cdot h_{t-1}$
```

Reducing search space. We propose three rules to prune the search space, removing sub-optimal and redundant schedules: i) Time-dependent phases must have symmetry across timesteps. As RNN computation is identical across timesteps, the fastest schedule for executing each timestep should also be identical. ii) If two consecutive phases are of the same type, then there must be a dependency between the two phases. If no dependency exists then this schedule is equivalent to another schedule where a single phase consists all MMs in both phases. iii) We compute time-independent phases before all dependent ones, as shown in Schedule 2 of Lst. 2. Having phases of the same type in consecutive order increases reuse of weights.

5.2. Data Locality Optimizations

DeepCPU improves data reuse within each phase and across phases through four techniques.

5.2.1 Fusion of MMs

DeepCPU fuses all possible MMs within each phase — *Two MMs can be fused into a single MM if they share a common input matrix.*

How to fuse? Consider two MMs, $MM1 : \mathbf{C1}[i1, j1] = \sum_{k1} \mathbf{A1}[i1, k1] \times \mathbf{B1}[k1, j1]$ and $MM2 : \mathbf{C2}[i2, j2] = \sum_{k2} \mathbf{A2}[i2, k2] \times \mathbf{B2}[k2, j2]$. W.l.o.g., assume $\mathbf{A1}[i1, k1] = \mathbf{A2}[i1, k1]$, as shared input matrix. The two MMs can be fused into a single one $MM12$ by concatenating $\mathbf{B1}$ and $\mathbf{B2}$, and $\mathbf{C1}$ and $\mathbf{C2}$ along the column, i.e., $\mathbf{C12}[i1, j12] = \sum_{k1} \mathbf{A1}[i1, k1] \times \mathbf{B12}[k1, j12]$ where $\mathbf{B12}[k1, j1] = \mathbf{B1}[k1, j1]$, $\mathbf{B12}[k2, J1 + j2] = \mathbf{B2}[k2, j2]$, and $\mathbf{C12}[i1, j1] = \mathbf{C1}[i1, j1]$, $\mathbf{C12}[i2, J1 + j2] = \mathbf{C2}[i2, j2]$ ($J1$ is the size of index $j1$).

Why fuse? Fusion improves data reuse. Consider using any GEMM implementation to execute $MM1$ and $MM2$ without fusion. While both $MM1$ and $MM2$ share a common input, GEMM is not aware of this reuse and could not take advantage of it. However, if we fuse them, this reuse is explicit in the MM and GEMM can exploit it to improve both performance and scalability.

5.2.2 Reuse-aware Parallelism Generator

Parallelism boosts compute capacity but may also increase data movement. This part discusses the relation of locality and parallelism, and our parallelism strategy.

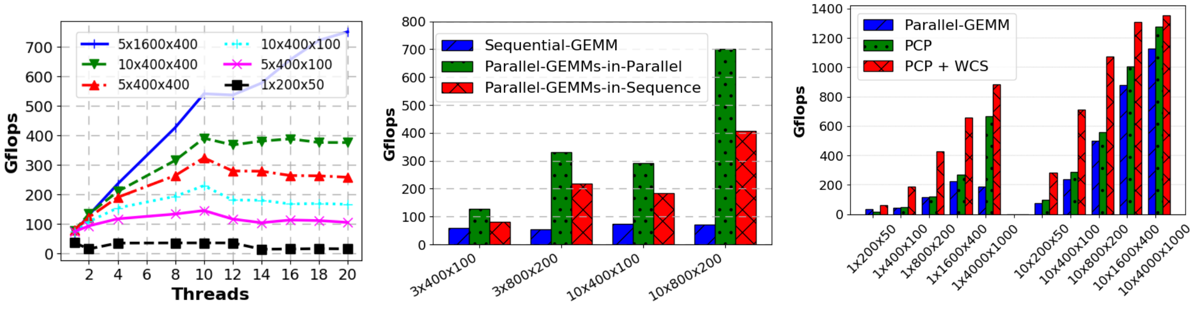
How to parallelize a single MM? Executing an MM with the maximum available parallelism is not always the best option for performance. As the parallelism increases, either the input or output must be replicated across multiple $L2$ private caches, thus increasing the total data movement. Once the level of parallelism reaches a certain threshold, the performance is limited by the data movement instead of the computational throughput. As shown in Fig. 3a, the MM performance degrades after certain parallelism. It is crucial to find the optimal level of parallelism instead of applying the common wisdom of using all available cores.

How to parallelize concurrent MMs? Multiple MMs within a phase do not have any dependencies. DeepCPU executes them as *Parallel-GEMMs-in-Parallel*, where multiple MMs are executed concurrently with each MM executing in parallel. For example, to compute two independent MMs, $M1$ and $M2$, on P cores, we run $M1$ and $M2$ in parallel, each using $P/2$ cores. This is in contrast with Parallel-GEMMs-in-Sequence, where we run $M1$ first using P cores followed by $M2$. Parallelizing an MM across multiple cores increases the data movement from $L3$ to $L2$ cache. In contrast, executing multiple MMs in parallel across multiple divided groups of cores allows each group to work on a unique MM without requiring data replication across them, improving data reuse while maintaining the same parallelism level. Fig. 3b shows empirical results. We run two independent and identical MMs with increased parallelism and report the best performance achieved. Parallel-GEMMs-in-Parallel significantly outperforms Parallel-GEMMs-in-Sequence.

How to optimize parallelism degree? Finding the optimal parallelism degree analytically is non-trivial as it depends on many architectural parameters. However, it is also not necessary in practice. DeepCPU applies Parallel-GEMMs-in-Parallel if a phase has multiple fused MMs. It then uses auto-tuning to identify the optimal parallelism for the phase quickly, as the number of cores on a modern multi-core CPU is less than two orders of magnitude and well-known RNN operators such as LSTMs/GRUs have at most two fused MMs per phase.

5.2.3 Private-Cache-Aware Partitioning (PCP)

We develop PCP, a novel private-cache-aware partitioning strategy for executing MMs across multicores to optimize $L2$ reuse within and across phases. PCP provides a principled method to optimize data movement with for-



(a)

(b)

(c)

Figure 3: For MMs of different sizes, (a) shows performance results by increasing the parallelism degree. (b) ompares Parallel-GEMMs-in-Parallel vs Parallel-GEMMs-in-Sequence, plus Sequential-GEMM as baseline. C) compares parallel-GEMM, PCP with and without weight-centric streamlining for running MMs.

mal analysis: For a given MM with parallelism degree P, we show PCP produces a P-partitioning of the computation space such that the total data movement between $L3$ and $L2$ cache is minimized. DeepCPU employs PCP to generate a locality-optimized schedule for each parallelism configuration without requiring to empirically calibrate different partitions and measure their performance.

Reuse within phases. Suppose an MM $\mathbf{C}[i,j] = \sum_k \mathbf{A}[i,k] \times \mathbf{B}[k,j]$ has P partitions, where X_i, X_j and X_k are the number of partitions along each of the i, j, k dimensions and $X_i \times X_j \times X_k = P$. We first derive the total data movement between $L3$ and $L2$ cache as a function of the partitions. This data movement depends on the relation between the size of the input and output matrices of the MM and the sizes of the $L3$ and $L2$ caches. For all RNNs of interest in serving scenario, we observe that the input matrix is much smaller than $L2$ cache, and the sum of all matrices fit in $L3$ cache. Under such conditions, we prove in Lemma 5.1 and Theorem 5.2 that the total data movement between $L3$ and $L2$ cache is equal to $X_j|\mathbf{A}| + X_i|\mathbf{B}| + 2X_k|\mathbf{C}|$. By choosing X_i, X_j, and X_k that minimizes this quantity, PCP obtains a parallel partitioning that maximizes data reuse from $L2$ cache.

Lemma 5.1. *The tight bound on data movement between a slow memory and a fast memory of size S for an MM* $\mathbf{C}[i,j] += \sum_k \mathbf{A}[i,k] \times \mathbf{B}[k,j]$ *is given by* $|\mathbf{A}| + |\mathbf{B}| + 2|\mathbf{C}|$, *when* $S \geq min(|\mathbf{A}|, |\mathbf{B}|, |\mathbf{C}|) + H + 1$. *Here we assume that the inputs and output matrices initially reside in the slow memory, and the final output must also reside in the slow memory. H is a constant not greater than* $\max(I,J,K)$, *where I, J and K are the sizes of indices i, j and k respectively.*

Proof. Lower bound: As both inputs and outputs originally reside in slow memory they must be read to fast memory at least once to compute the MM. After computation, the output must be written to slow memory at least once. That gives $|\mathbf{A}| + |\mathbf{B}| + 2|\mathbf{C}|$ as a lower bound.

Upper bound: W.l.o.g., assume $\mathbf{A}$ fits in S. Lst. 3 shows a schedule where the total data movement between slow and fast memory is given by $|\mathbf{A}| + |\mathbf{B}| + 2|\mathbf{C}|$, when $S \geq |\mathbf{A}| + K + 1$. Note H (=K) is a (small) buffer space to hold a single column of $\mathbf{B}$ during the computation. □

Listing 3: MM Schedule that achieves data movement of $|\mathbf{A}| + |\mathbf{B}| + 2|\mathbf{C}|$ when $S \geq |\mathbf{A}| + K + 1$

```
//C[i,j] = ∑k A[i,k] × B[k,j]
Load A[*,*] in A_buf //MemReq = |A|
for j
  Load B[*,j] in B_buf //MemReq = K
  for i
    Load C[i,j] in c //MemReq  = 1
    for k
      c += A_buf[i,k] × B_buf[k]
    Store c in C[i,j]
```

Theorem 5.2. *Consider P cores on a CPU, and an MM* $\mathbf{C}[i,j] += \sum_k \mathbf{A}[i,k] \times \mathbf{B}[k,j]$, *where* $|\mathbf{A}| + |\mathbf{B}| + |\mathbf{C}| \leq |L3Cache|$ *and* $min(|\mathbf{A}|, |\mathbf{B}|, |\mathbf{C}|) + H + 1 \leq |L2Cache|$. *H is a constant not greater than* $max(I,J,K)$, *where I, J and K are the sizes of indices i, j and k. For a P-way partitioning* $\langle X_i, X_j, X_k \rangle$ *where* $X_i \times X_j \times X_k = P$, *a tight bound on the data movement between L3 and L2 cache is given by* $X_j|\mathbf{A}| + X_i|\mathbf{B}| + 2X_k|\mathbf{C}|$.

Proof. Each of the partitions given by $\langle X_i, X_j, X_k \rangle$ is an MM of size $\frac{I}{X_i} \times \frac{J}{X_j} \times \frac{K}{X_k}$. From Lemma. 5.1 we see that a tight bound on the data movement between $L3$ cache and $L2$ cache for each of these sub-MMs is given by $\frac{I \times K}{X_i \times X_k} + \frac{K \times J}{X_k \times X_j} + 2\frac{I \times J}{X_i \times X_j}$. Thus the total data movement for all partitions is given by $X_i \times X_j \times X_k \times (\frac{I \times K}{X_i \times X_k} + \frac{K \times J}{X_k \times X_j} + 2\frac{I \times J}{X_i \times X_j}) = X_j|\mathbf{A}| + X_i|\mathbf{B}| + 2X_k|\mathbf{C}|$. □

Reuse across phases. PCP so far maximizes the data reuse by considering each phase independently. However, identical time-dependent phases (TDPs) across a sequence have data reuse between them. For each MM in these phases, weight matrices stay the same. We extend PCP to exploit the reuse in weights across phases.

For a given P-partitioning strategy $\langle X_i, X_j, X_k \rangle$, the weight matrix $\mathbf{B}$ is divided into blocks of size $\frac{|\mathbf{B}|}{X_j \times X_k}$. If this block fits in $L2$ cache of an individual core, then it will not be evicted from $L2$ cache for the entire computation sequence as long as the mapping between the MM partitions and the compute cores does not change. In such cases, denoting the sequence length of RNN as *seq_len*, the total data movement is given by

$$seq\_len \times (X_j|\mathbf{A}| + 2X_k|\mathbf{C}|) + X_i|\mathbf{B}|$$

as the weight matrix **B** needs to be read only once from $L3$ cache at the first time step. In general, total data-movement between $L3$ and $L2$ caches is calculated as

$$\begin{cases} seq\_len \times (X_j|\mathbf{A}| + 2X_k|\mathbf{C}|) + X_i|\mathbf{B}| & \text{if } \frac{|\mathbf{B}|}{X_j * X_k} \leq |L2| \\ seq\_len \times (X_j|\mathbf{A}| + X_i|\mathbf{B}| + 2X_k|\mathbf{C}|) & \text{if } \frac{|\mathbf{B}|}{X_j * X_k} > |L2| \end{cases}$$

By minimizing this piecewise function, we maximize the data reuse across a sequence. In practice, it is not necessary for a block of the weight matrices to fit entirely in $L2$ cache. As long as the block is not much larger than $L2$ cache, we can still get partial reuse.

5.2.4 Weight-Centric Streamlining (WCS)

WCS is our implementation to enable full-fledged PCP, supporting reuse of weight matrices across TDPs. For a given parallelism degree, PCP produces a partitioning such that the weights required to compute the partition fit in $L2$ cache of a single core (when possible), allowing the weights to be reused from $L2$ cache across TDPs, without being evicted. However, to ensure this reuse, the computation must be conducted at where the weights are, i.e., the mapping between parallel partitions and the cores that execute them must not change across TDPs.

To this end, we use OpenMP [24] to create a parallel region that spans the entire RNN sequence of computation. The parallelism degree is equal to the max parallelism degree among all phases in the schedule. Each thread in the parallel region is responsible for executing at most a single parallel partition during each phase. Some threads may remain idle during phases where the parallelism degree is less than the number of threads. Each thread ID is mapped to a unique partition ID, and this mapping is identical across TDPs. In essence, we alternate the order of the sequence loop and the parallel region such that the sequence loop is inside the parallel region, shown as ParallelOuterRNN in Lst. 4.

Listing 4: Parallel Outer vs Parallel Inner RNN

```
ParallelOuterRNN(intput_sequence , output)
  #pragma omp parallel
    int id = omp_get_thread_num()
    for t in intput_sequence:
      ComputeRNNOuterParallel(id, t, output)

ParallelInnerRNN(intput_sequence,output)
  for t in intput_sequence:
    #pragma omp parallel
      int id = omp_get_thread_num()
      ComputeRNNInnerParallel(id, t, output)
```

This has two major advantages over creating parallel regions inside the sequence loop as ParallelInnerRNN, i) it allows easy pinning of each MM parititon to a particular core across RNN steps. In OpenMP, threads in each parallel region have their local thread IDs starting from 0. A unique mapping between this local thread ID and the global thread ID is not guaranteed across multiple parallel regions separated in time. Thread affinity settings allow binding global thread IDs to cores or hyperthreads, but not local thread IDs. By creating a single parallel region, we create a unique mapping between a local thread ID and the global thread ID throughout the computation, which ensures that an MM partition is always executed on the same core across the entire sequence. ii) It reduces the overhead of creating parallel regions. Instead of opening and closing parallel regions during each step of the RNN sequence, we only create a parallel region once for the entire computation.

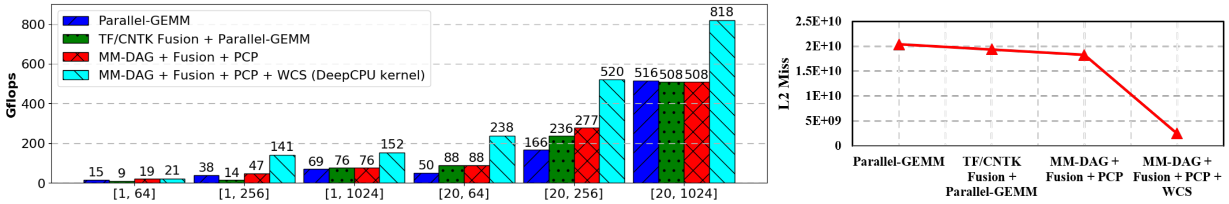
Fig. 3c compares performance of running a sequence of parallel-GEMM and PCP with/without WCS for varied sizes of MMs. The latter two consistently outperform the former, but the full benefit of PCP (across phases) is realized only when used together with WCS.

5.3. Performance Impact of Optimization Techniques

We compare four implementations using different LSTM configurations: i) Parallel-GEMM(baseline): Runs each step of LSTM as 8 MMs in sequence, and each MM is executed with Intel-MKL parallel-GEMM. ii) TensorFlow/CNTK Fusion: the fused MM (as Lst. 1) is executed using Intel-MKL parallel-GEMM. iii) MM-DAG+Fusion+PCP: All optimizations in DeepCPU except WCS. iv) DeepCPU Kernel: All aforementioned optimizations.

Results. TensorFlow/CNTK Fusion has roughly the same performance as baseline. MM-DAG+Fusion+PCP is as good as or better than both of them. It searches for the fused phased schedules including TensorFlow/CNTK fusion, as well as those that increase reuse by fusing across time steps. It also applies PCP for better partitioning. However, it does not ensure that MMs sharing same weights are mapped to the same core. In contrast, DeepCPU kernel is often much faster, particularly for small batch sizes where the reuse is small within a single phase and reuse across TDPs must be exploited for better performance. Even for larger batch size with the input/hidden dimension 256 and 1024, where the total size of the weight matrices is larger than the $L2$ cache but individual weight blocks fit in $L2$ cache, DeepCPU kernel offers good speedup by enabling reuse of weights across TDPs.

Performance counters. We measure the amount of data movement from $L2$ to $L3$ through $L2$ cache misses using *L2_RQSTS.ALL_DEMAND_MISS* counter in Intel® VTune™ Amplifier [2]. Fig.4b shows DeepCPU significantly reduces $L2$ cache misses (by 8 times), verifying its effectiveness on locality optimization.



(a) (b)

Figure 4: (a) Performance of LSTMs (in the form of [batch size, input/hidden dimension]) with different optimization techniques. (b) $L2$ cache misses for config [20, 256]. Measured at $sequence\_length = 100$ with 2000 iterations.

Search space size. DeepCPU finds the optimal execution schedule with just a few hundred calibration runs. In the example of LSTM, we search approximately $P \times Q$ configurations by generating $P = \#cores$ parallelism choices, and Q phased schedules that satisfies all three pruning criteria described in Sec. 5.1. For LSTMs, $Q < 20$, which can be verified by enumerating all such valid schedules. Per parallelism choice, PCP identifies optimized partitioning analytically (e.g., integer programming) without requiring additional empirical exploration, greatly saving search space. This search/calibration process is often called once during model construction, and then the optimized schedule is repeatedly used for serving upcoming user requests.

6. Evaluation

We compare DeepCPU with RNN implementations from state-of-the-art DL frameworks: DeepCPU is an order of magnitude faster than TensorFlow and CNTK for a wide range of RNN models and configurations on CPUs. DeepCPU also outperforms GPU implementations significantly for most of the cases. Furthermore, we test DeepCPU on real-world applications used in production: it transforms these models from impossible to ship due to latency violation to well-fitting SLA while also saving millions of dollars in infrastructure cost.

Hardware. Our evaluation is conducted on a server with two 2.20 GHz Intel Xeon E5-2650 V4 processors, each of which has 12-core (24 cores in total) with 128GB RAM, running 64-bit Linux Ubuntu 16.04. The peak Gflops of the CPU is around 1.69Tflops. The server has one Nvidia GeForce GTX TITAN X which is used for measuring RNN performance on GPU.

6.1. RNN Performance Comparison

Workload. We evaluate LSTM/GRU by varying input dimension, hidden dimension, batch size, and input sequence length to cover a wide range of configurations.

Comparison frameworks. There are many DL frameworks such as TensorFlow [13], CNTK [52], Caffe2 [37], Torch [41], Theano [17], and MXNet [19] that support RNNs on CPUs and GPUs. We compare DeepCPU with TensorFlow and CNTK. We choose TensorFlow because it is adopted widely. We use TensorFlow version 1.1.0 with Accelerated Linear Algebra (XLA) compiler, optimizing pointwise kernels, and with Intel Math Kernel Library (MKL) for efficient matrix operations. We let TensorFlow pick appropriate degrees for inter-op and intra-op parallelism. We choose CNTK since a recent study showed that it achieves good performance on RNNs [55]. CNTK also uses MKL and sets the number of threads equal to the number of cores for MMs by default. On GPUs, we evaluate TensorFlow, CNTK and a highly optimized cuDNN implementation [14, 20].

Speedup on CPUs. Table 1 presents the execution time and speedup results of DeepCPU, in comparison to TensoFlow and CNTK on CPUs, covering a wide range of RNN model sizes. The first four columns describe the specification of RNNs: input dimension, hidden dimension, batch size, and sequence length. Both absolute execution time and speedup are reported. Speedup is measured as the ratio between the execution times of TensorFlow (or CNTK) versus DeepCPU, e.g., a value of 2 indicates that DeepCPU is 2 times faster. To make reliable measurement, we run each config 2000 times and report the average. The results show that *DeepCPU significantly and consistently outperforms TensorFlow and CNTK, with speedup in the range of 3.7 to 93 times, and average speedup of 18X among all tested configurations.*

Next, we conduct an in-depth performance comparison, showing how model parameters affect the results, on both CPU and GPU, across 6 implementations.

Varying input/hidden dimension. Fig. 5a reports the execution time and Gflops of LSTMs with varying input/hidden dimension from 32 to 1024. This is the range of dimension size commonly observed from RNN models in practice. Here we choose batch size of 1 to represent a common case in serving. As expected, the execution time for all implementations increases with the increase in dimension size. However, compared to all implementations, DeepCPU always has the shortest execution time and the highest Gflops on both CPUs and GPUs for all sizes. Note that the y-axis of the execution time is in log-scale, so the actual gap is larger than it appears. DeepCPU is more than an order of magnitude faster than both TensorFlow and CNTK on CPUs. On GPUs, DeepCPU has significantly higher performance when the dimension size is small or medium (e.g., less than 256). As the dimension size gets larger, this perfor-

Model parameters				LSTM exec. time (ms)			GRU exec. time (ms)			LSTM speedup		GRU speedup	
input	hidden	batch	seq. len.	TF	CNTK	DeepCPU	TF	CNTK	DeepCPU	TF	CNTK	TF	CNTK
64	64	1	100	7.3	25	0.31	8	25	0.7	26	81	11	36
256	64	1	100	10	27	0.29	9.6	26	0.58	34	93	17	45
1024	64	1	100	19	25	0.42	16	27	0.69	45	60	23	39
64	256	1	100	21	23	0.62	17	30	0.79	34	37	22	38
64	1024	1	100	180	30	6.5	110	37	6.4	28	4.6	17	5.8
1024	1024	1	100	460	33	11	190	40	8.4	42	3	23	4.8
256	256	1	1	0.96	1.1	0.069	0.89	1	0.053	14	16	17	19
256	256	1	10	3.4	2.9	0.16	2.9	3.4	0.14	21	18	21	24
256	256	1	100	28	21	0.74	22	25	0.9	38	28	24	28
64	64	10	100	20	47	1.1	18	43	1.1	18	43	16	39
64	64	20	100	27	74	1.5	25	88	1.5	18	49	17	59
256	256	10	100	51	62	4.4	34	66	3.7	12	14	9.2	18
256	256	20	100	58	91	6.4	51	100	5.4	9.1	14	9.4	19
1024	1024	10	100	400	180	42	280	170	36	9.5	4.3	7.8	4.7
1024	1024	20	100	540	250	68	380	230	60	7.9	3.7	6.3	3.8

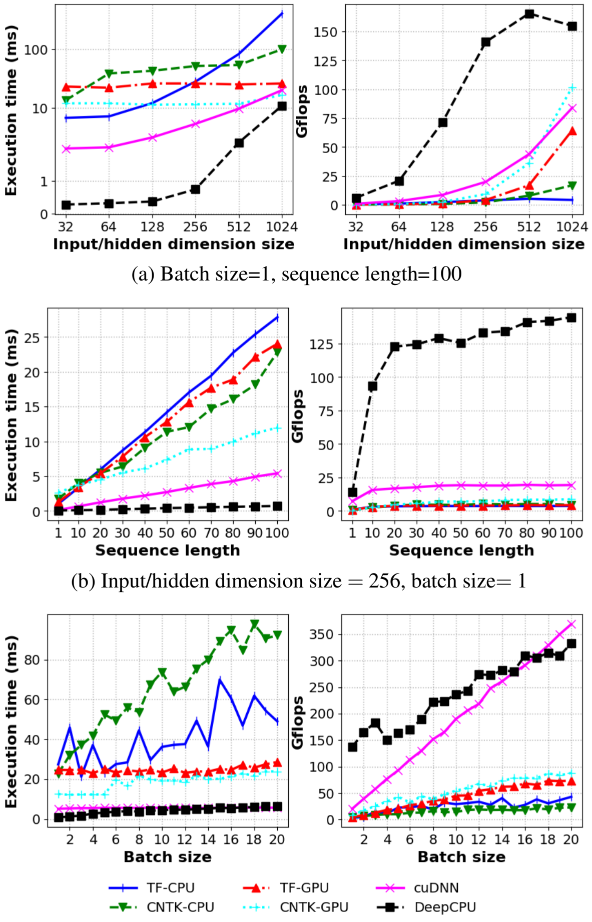
Table 1: Execution times and speedups of LSTMs and GRUs, comparing DeepCPU, TensorFlow and CNTK on CPU.

mance gap decreases due to increase in parallelism that allows for an increasing number of GPU cores to kick in. On the other hand, the CPU Gflops plateaus after dimension size of 512.

Varying sequence length. Fig. 5b shows the performance impact of varying input sequence lengths from 1 to 100. As the sequence length increases, the execution time of all implementations except DeepCPU increases almost linearly, or equivalently, their Gflops stays constant. DeepCPU, however, has a sharp jump in performance when sequence length increases from 1 to 10. It demonstrates that DeepCPU exploits data reuse across steps: when sequence length > 1, later steps reuse the weights from the first step, increasing Gflops. Once the sequence length becomes larger than 10, the increase in reuse per flop is marginal. Thus, the Gflops curve is relatively flat when sequence length grows from 20 to 100.

Varying batch size. As shown in Fig. 5c, among all CPU implementations, DeepCPU performs and scales the best with increasing batch size. Among GPU implementations, cuDNN performs significantly better than TensorFlow and CNTK. Comparing DeepCPU with cuDNN, the best CPU versus GPU implementation, DeepCPU is better with small and moderate batch size (< 15) and cuDNN is better with large batch sizes. This crossover is expected. However, as discussed earlier, batch size is often rather small for serving scenarios due to the stringent latency SLA.

The GPU implementation in existing framework such as TF-GPU has worse performance than cuDNN. This is because TF-GPU and cuDNN do not use the same underlying implementation. In the case of LSTMs, TensorFlow constructs the LSTM operator as a composition of matrix multiplications and activation functions. A single LSTM operator produces hundreds of nodes in the TensorFlow computation graph. While some of these nodes are computed on the GPUs (for example matrix multiplication using cuBLAS), transferring tensors among nodes incurs quite significant overhead. In contrast, cuDNN



(a) Batch size=1, sequence length=100

(b) Input/hidden dimension size = 256, batch size= 1

(c) Input/hidden dimension size = 256, sequence length= 100

Figure 5: LSTM execution time and Gflops with varying input/hidden dimension, sequence length and batch size.

implementation is a single highly optimized kernel invocation for the entire sequence of the LSTM computation.

6.2. Serving Real-World RNN-Based Models

We evaluate DeepCPU on serving three real-world models. Table 2 provides their RNN specifications.

What's inside DeepCPU? DeepCPU focuses on RNN families and supports LSTM/GRU cell, LSTM/GRU sequence, bidirectional RNN, and stacked RNN networks. It includes fundamental building blocks such as efficient matrix multiplication kernels, activation functions, as well as common deep learning layers such as highway network [57], max-pooling layer [40], multilayer perceptron [50], variety of attention layers [39, 53], and sequence-to-sequence decoding with beam search [59].

Converting trained models into DeepCPU. DeepCPU focuses on serving, and we take a two-step approach to convert trained models (e.g., from TensorFlow/CNTK) to use it. 1) Replace the RNN part(s) of the original model using DeepCPU APIs. In this paper, we implement all three models using DeepCPU C++ APIs. The engineering work is manageable as our library contains many reusable and common components for building neural networks. A more automated way is to integrate our library with an existing framework, which we consider as future work. 2) Port the weights of the trained model to initialize the DeepCPU model instances.

Text similarity (TS). TS measures semantic similarity between texts [45]. It is widely used for grading machine translation results, detecting paraphrase, and ranking query document relevance. It uses bidirectional GRUs to encode text inputs (e.g., sentences) into semantic vectors and measures their relevance with cosine similarity. The GRU computation dominates the performance of the model. The first row in Fig. 6 shows that with DeepCPU, TS runs 12X faster than TensorFlow on CPUs and 15X faster than TensorFlow on GPUs.

Attention sum reader (ASR). ASR extracts single token answer from a given context and can be used for online question and answering [39]. The model uses bidirectional GRU to encode both query and context into semantic vectors and performs reasoning steps to figure out which token in the context is the answer. Fig. 6 shows that DeepCPU reduces ASR serving latency from more than 100ms to less than 10ms, a more than 10X speedup over TensorFlow on CPUs and GPUs.

Bidirectional attention flow model (BiDAF). BiDAF is a high-ranked model on SQuAD reading comprehension competition list [11] for question and answering [53]. It has a hierarchical structure composed of five neural network layers. Among them, three are LSTM-based (Table 3). Fig. 6 shows that DeepCPU reduces the execution time of BiDAF from more than 100ms to less than 5ms, achieving more than 20*x* speedup against TensorFlow. Table 3 lists the execution time breakdown across layers after the optimization: DeepCPU significantly decreases the execution time of LSTM-based layers.[1]

Correctness. We use TensorFlow for correctness verification: DeepCPU always produces prediction results matching those generated by TensorFlow.

Hardware choice. While not reported in the paper, we have tried DeepCPU on a few different SKUs and processor generations. We have found significant performance improvements even on Haswell and Ivy Bridge generations. The techniques are effective as long as the model is not significantly larger than $L3$ cache of the hardware. DeepCPU also provides additional performance boost from weight-centric streamlining when the weight matrices fit in $L2$ caches of multiple cores.

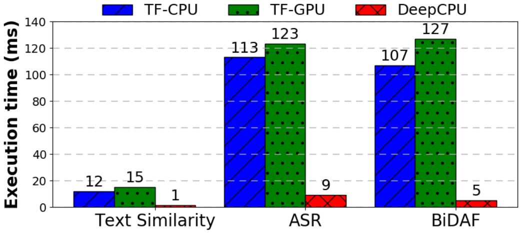


Figure 6: Execution time of TS, ASR, and BiDAF.

Meeting latency SLA with significant cost savings. Besides greatly reducing latency to meet SLA, DeepCPU significantly improves efficiency and reduces serving cost. Take TS model as an example, which is used for ranking query and document pair at our search services. The latency SLA is 6ms and 33 selected documents are ranked for each query. The original TensorFlow model takes 12ms to serve a single ⟨query, document⟩ pair on one CPU machine, violating latency SLA and unable to ship. DeepCPU not only reduces the latency to meet SLA, but more importantly, as shown in Table 4, it only takes 5.6ms to serve a query and *all of its 33 document* on the same machine. *DeepCPU achieves more than 60x throughput gain (i.e.,* $12 \times 33/5.6$*).* Our large-scale search service answers tens of thousands of requests per second, and would originally require more than 10K machines for hosting this model. DeepCPU reduces it to a couple hundred, saving millions of dollars of infrastructure cost just for this model alone.

7. Related Work

DL acceleration library. The closest work to DeepCPU are cuDNN [14, 20] and MKL-DNN [4], which are libraries for accelerating DL frameworks. CuDNN is a GPU library mainly designed for maximizing training throughput, and its performance can be limited by insufficient parallelism when the model size and batch size are small. Other work on optimizing RNNs also target

[1] We also optimized embedding and attention layer to improve end-to-end latency, where the details are beyond the scope of the paper.

Model	RNN parameters
Text similarity [45]	*–input 200 –hidden 512 –source_length 20 –target_length 20 –batch_size 1*
ASR [39]	*–input 200 –hidden 256 –question_length 20 –context_length 100 –context_batch_size 10*
BiDAF [53]	*Phrase embedding: –input 50 –hidden 100 –question_length 15 –context_length 100 –context_batch_size 1* *Modeling layer (stackd LSTM): –input 800 –hidden 100 –context_length 100 –context_batch_size 1* *Output layer: –input 1400 –hidden 100 –context_length 100 –context_batch_size 1*

Table 2: The description of model parameters of RNNs used in real-world models. Sequence lengths refer to maximum sequence length, and both TensorFlow and DeepCPU support variable sequence lengths.

	TF on CPUs	DeepCPU
Embedding + highway	0.69	0.84
Phrase embedding (LSTMs)	13	0.23
Attention layer	13	1.30
Modeling layer (LSTMs)	31	0.90
Output layer (LSTMs)	49	1.50
Total	107	4.77

Table 3: BiDAF execution time (millisecond) per layer.

	T@10	T@15	T@20	T@33
Embedding	0.28	0.24	0.24	0.36
RNN	2.20	2.60	3.40	5.20
Cosine similarity	0.04	0.04	0.04	0.04
Total	2.50	2.90	3.60	5.60

Table 4: Text similarity model execution time where T@K reports execution time of ⟨query, K documents⟩.

GPUs [26, 30, 64]. On CPUs, MKL-DNN is a C/C++ library from Intel to boost DL model performance on Intel architecture, but it only supports convolutional neural networks and has no support for RNNs yet. Other work on multi-core CPUs is similar, targeted more towards CNNs, fully connected neural networks, etc [43, 49, 61]. Some DeepCPU optimizations (e.g., parallelization, fusion) can be generalized to these other networks, whereas optimizations like WCS are more specialized to RNNs.

Compiler and runtime optimizations. There has been work on optimizing DL model performance through compile-time and runtime strategies. Many of them use static analysis to find pipelined operations that can be fused together for improved performance, such as XLA [10], Weld [47], and TensorRT [5]. The compile-time and runtime strategies of these systems are not designed for global optimization of complex structures like RNN sequences. Halide is a domain specific language and compiler to optimize image processing pipeline [48], conveniently separating algorithms with schedules. It is not specially designed for RNN type of recurrent computation, and optimizations such as the weight-centric streamlining cannot be supported easily. It is also hard for its autotuner to search the space efficiently without domain-specific pruning and partitioning methods.

Model deployment. TensorFlow Serving [9] and Clipper [23] are two serving platforms for deploying and serving machine learning models on production systems. Both support caching inference results and batching individual inference requests for better performance. Clipper selects from multiple models to balance latency with accuracy. Our work and these model deployment platforms complement each other: while they focus on the deployment process for serving requests, our library focus on optimizing the inference time of a model itself.

Hardware accelerators. Apart from CPU and GPU, researchers and practitioners are also looking into specialized hardware such as FPGA [42, 46, 54] and ASIC [32, 38], which often require expert hardware designers and long development cycles to obtain high performance. They are not yet widely available commercially.

Model simplification and compression. Existing work shows many model simplification techniques [33, 36, 63] such as sparsifying and quantization that could reduce computation time and space with a small accuracy tradeoff. Co-designing these model optimizations together with system optimizations like those in DeepCPU could present new opportunities to boost performance further.

8. Conclusion

The paper unravels the mystery of poor RNN performance on existing DL frameworks — low data reuse — and develops optimization schemes to reduce latency and improve efficiency of RNN serving. Powered by the new techniques and search strategy, DeepCPU, our serving library on CPUs, improves performance by an order of magnitude, compared with existing work. It transforms many RNN models from non-shippable to shippable with great latency and cost improvement in production.

9. Acknowledgments

We thank great collaborators for their support of DeepCPU, including Junhua Wang, Rangan Majumder, Saurabh Tiwary, Yuesheng Liu, Qifa Ke, Bruce Zhang, Chandu Thekkath, Yantao Li, Jason Li, Fang Liu, and their team. We thank Bin Hu, Elton Zheng, Aniket Chakrabarti, Ke Deng, Doran Chakraborty, Guenther Schmuelling, Stuart Schaefer, Michael Carbin, Olatunji Ruwase, and Sameh Elnikety. The usability and functionality of DeepCPU has been greatly expanded by incorporating their feedback. We thank Kathryn McKinley for reading a previous version of this paper and providing feedback. We thank our shepherd, Deniz Altinbuken, for helping improving the presentation quality of the work.

References

[1] Eigen C++ Library for Linear Algebra. http://eigen.tuxfamily.org.

[2] Intel VTune Amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[3] Intel(R) Math Kernel Library. https://software.intel.com/en-us/mkl.

[4] Intel(R) Math Kernel Library for Deep Neural Networks. https://github.com/01org/mkl-dnn.

[5] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[6] Openblas. www.openblas.net/.

[7] Recurrent Neural Network Tutorial. http://www.wildml.com/2015/09/.

[8] Stream Benchmark. http://www.cs.virginia.edu/stream/ref.html.

[9] TensorFlow Serving. https://www.tensorflow.org/serving/.

[10] The Accelerated Linear Algebra Compiler Framework. https://www.tensorflow.org/performance/xla/.

[11] The Stanford Question Answering Dataset (SQuAD) leaderboard. https://rajpurkar.github.io/SQuAD-explorer/.

[12] Understanding LSTM Networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 265–283, 2016.

[14] Jeremy Appleyard, Tomás Kociský, and Phil Blunsom. Optimizing Performance of Recurrent Neural Networks on GPUs. arXiv preprint arXiv:1604.01946, 2016.

[15] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. arXiv preprint arXiv:1409.0473, 2014.

[16] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A Neural Probabilistic Language Model. *The Journal of Machine Learning Research*, 3:1137–1155, 2003.

[17] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-farley, and Yoshua Bengio. Theano: A CPU and GPU Math Compiler in Python. In *Proceedings of the 9th Python in Science Conference*, pages 3–10, 2010.

[18] Danqi Chen, Jason Bolton, and Christopher D. Manning. A Thorough Examination of the CNN/Daily Mail Reading Comprehension Task. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL '16*, 2016.

[19] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv preprint arXiv:1512.01274, 2015.

[20] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. arXiv preprint arXiv:1410.0759, 2014.

[21] Kyunghyun Cho, Bart van Merriënboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP' 14*, pages 1724–1734, 2014.

[22] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. arXiv preprint arXiv:1412.3555, 2014.

[23] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI '17*, pages 613–627, 2017.

[24] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[25] Bhuwan Dhingra, Hanxiao Liu, Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. Gated-Attention Readers for Text Comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL '17*, pages 1832–1846, 2017.

[26] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Y. Hannun, and Sanjeev Satheesh. Persistent RNNs: Stashing Recurrent Weights On-Chip. In *Proceedings of the 33nd International Conference on Machine Learning, ICML '16*, pages 2024–2033, 2016.

[27] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing Web Latency: The Virtue of Gentle Aggression. In *Proceedings of the ACM Conference of the Special Interest Group on Data Communication*, SIGCOMM '13, pages 159–170, 2013.

[28] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. 1990.

[29] Kazushige Goto and Robert A. van de Geijn. Anatomy of High-performance Matrix Multiplication. *ACM Transactions on Mathematical Software*, 34(3):12:1–12:25, May 2008.

[30] Alex Graves. RNNLIB: A Recurrent Neural Network Library for Sequence Learning Problems. https://github.com/szcom/rnnlib.

[31] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech Recognition with Deep Recurrent Neural Networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '13*, pages 6645–6649, 2013.

[32] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 243–254, 2016.

[33] Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *International Conference on Learning Representations, ICLR '16*, 2016.

[34] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep Speech: Scaling up end-to-end speech recognition. arXiv preprint arXiv:1412.5567, 2014.

[35] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[36] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks. In *Advances in Neural Information Processing Systems 29*, pages 4107–4115. 2016.

[37] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, 2014.

[38] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, 2017.

[39] Rudolf Kadlec, Martin Schmid, Ondrej Bajgar, and Jan Kleindienst. Text Understanding with the Attention Sum Reader Network. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL '16*, 2016.

[40] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012*, NIPS '12, pages 1106–1114, 2012.

[41] Nicholas Léonard, Sagar Waghmare, Yang Wang, and Jin-Hwa Kim. rnn : Recurrent Library for Torch. arXiv preprint arXiv:1511.07889, 2015.

[42] Sicheng Li, Chunpeng Wu, Hai Li, Boxun Li, Yu Wang, and Qinru Qiu. FPGA Acceleration of Recurrent Neural Network Based Language Model. In *Proceedings of the 23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '15, pages 111–118, 2015.

[43] Ian Masliah, Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, Marc Baboulin, Joël Falcou, and Jack J. Dongarra. High-Performance Matrix-Matrix Multiplications of Very Small Matrices. In *22nd International Conference on Parallel and Distributed Computing, Euro-Par '16*, pages 659–671, 2016.

[44] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent Neural Network Based Language Model. In *11th Annual Conference of the International Speech Communication Association, INTERSPEECH '10*, pages 1045–1048, 2010.

[45] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic Regularities in Continuous Space Word Representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, NAACL-HLT' 13, pages 746–751, 2013.

[46] Eriko Nurvitadhi, Jaewoong Sim, David Sheffield, Asit K. Mishra, Krishnan Srivatsan, and Debbie Marr. Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. In *26th International Conference on Field Programmable Logic and Applications, FPL '16*, pages 1–4, 2016.

[47] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A common runtime for high performance data analytics. CIDR '17, 2017.

[48] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, 2013.

[49] Samyam Rajbhandari, Yuxiong He, Olatunji Ruwase, Michael Carbin, and Trishul Chilimbi. Optimizing CNNs on Multicores for Scalability, Performance and Goodput. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 267–280, 2017.

[50] Dennis W Ruck, Steven K Rogers, Matthew Kabrisky, Mark E Oxley, and Bruce W Suter. The multilayer perceptron as an approximation to a Bayes optimal discriminant function. *IEEE Transactions on Neural Networks*, 1(4):296–298, 1990.

[51] Hojjat Salehinejad, Julianne Baarbe, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. Recent Advances in Recurrent Neural Networks. arXiv preprint arXiv:1801.01078, 2018.

[52] Frank Seide and Amit Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, 2016.

[53] Min Joon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional Attention Flow for Machine Comprehension. arXiv preprint arXiv:1611.01603, 2016.

[54] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From High-Level Deep Neural Models to FPGAs. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '16*, pages 1–12, 2016.

[55] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking State-of-the-Art Deep Learning Software Tools. In *7th International Conference on Cloud Computing and Big Data, CCBD 2016*, pages 99–104, 2016.

[56] Daniele G. Spampinato and Markus Püschel. A Basic Linear Algebra Compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 23:23–23:32, 2014.

[57] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway Networks. arXiv preprint arXiv:1505.00387, 2015.

[58] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014*, pages 3104–3112, 2014.

[59] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014*, NIPS '14, pages 3104–3112, 2014.

[60] AJ Van der Poorten. Continued fraction expansions of values of the exponential function and related fun with continued fractions. *Nieuw Archief voor Wiskunde*, 14:221–230, 1996.

[61] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on CPUs. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS '11*, 2011.

[62] Oriol Vinyals and Quoc V. Le. A Neural Conversational Model. arXiv preprint arXiv:1506.05869, 2015.

[63] Wei Wen, Yuxiong He, Samyam Rajbhandari, Wenhan Wang, Fang Liu, Bin Hu, Yiran Chen, and Hai Li. Learning Intrinsic Sparse Structures within Long Short-term Memory. In *the Sixth International Conference on Learning Representations, ICLR '18*, 2018.

[64] Felix Weninger, Johannes Bergmann, and Björn Schuller. Introducing CURRENNT: The Munich Open-source CUDA Recurrent Neural Network Toolkit. *Journal of Machine Learning Research*, 16(1):547–551, 2015.

[65] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, April 2009.

[66] Geoffrey Zweig, Chengzhu Yu, Jasha Droppo, and Andreas Stolcke. Advances in all-neural speech recognition. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '17*, pages 4805–4809, 2017.